Московский авиационный институт
(Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа № 6

Тема: Основы работы с коллекциями: аллокаторы

Студент: Пивницкий Даниэль Сергеевич
Группа: 80-206
Преподаватель: Чернышов Л.Н.
Дата:
Оценка:

Москва, 2020

1. Постановка задачи

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Классы должны иметь публичные поля. Фигуры являются фигурами вращения, т.е. равносторонние (кроме трапеции и прямоугольника). Для хранения координат фигур необходимо использовать шаблон std::pair.

2. Описание программы

Аллокатор, совместимый со стандартными функциями, описан в allocator.h и используется коллекцией stack.

3. Набор тестов

test_1

```
m
1
1 2 3 4 6 2 3 0
5
1
0 0    1 0    2 0    2 1
5
2
2
2 2    1 2    0 2    0 1
5
4
2
5
q
```

test_2

```
m
1
1 3 4 6  2 3  0 5
5
1
0 0    1 0    2 0    2 1
5
4
1
```

4
0
5
q

4. Результаты выполнения тестов

choose option (m to open man, q to quit)

m

1) push new element into queue

2) insert element into chosen position

3) pop element from the queue

4) delete element from the chosen position

5) print queue

6) count elements with area less then chosen value

choose option (m to open man, q to quit)

1

enter rectangle (have to enter dots consequently):

1 2

3 4

6 2

3 0

choose option (m to open man, q to quit)

5

(1 2), (3 4), (6 2), (3 0)

choose option (m to open man, q to quit)

1

enter rectangle (have to enter dots consequently):

0 0 1 0 2 0 2 1

choose option (m to open man, q to quit)

5

(1 2), (3 4), (6 2), (3 0)

(0 0), (1 0), (2 0), (2 1)

choose option (m to open man, q to quit)

2

enter position to insert to: 2

enter rectangle: 2 2 1 2 0 2 0 1

choose option (m to open man, q to quit)

5

(1 2), (3 4), (6 2), (3 0)

(2 2), (1 2), (0 2), (0 1)

(0 0), (1 0), (2 0), (2 1)

choose option (m to open man, q to quit)

4

enter position to delete: 2

choose option (m to open man, q to quit)

5

(1 2), (3 4), (6 2), (3 0)

(0 0), (1 0), (2 0), (2 1)

choose option (m to open man, q to quit)

q

C:\Users\Daniel\Desktop\oop_exercise_05-master\out\build\x64-Debug\oop_exe

rcise_05.exe (процесс 14724) завершил работу с кодом 0.

choose option (m to open man, q to quit)

m

1) push new element into queue

2) insert element into chosen position

3) pop element from the queue

4) delete element from the chosen position

5) print queue

6) count elements with area less then chosen value

choose option (m to open man, q to quit)

1

enter rectangle (have to enter dots consequently):

1 3 4 6 2 3 0 5

choose option (m to open man, q to quit)

5

(1 3), (4 6), (2 3), (0 5)

choose option (m to open man, q to quit)

1

enter rectangle (have to enter dots consequently):

0 0 1 0 2 0 2 1

choose option (m to open man, q to quit)

5

(1 3), (4 6), (2 3), (0 5)

(0 0), (1 0), (2 0), (2 1)

choose option (m to open man, q to quit)

4

enter position to delete: 1

choose option (m to open man, q to quit)

4

enter position to delete: 0

choose option (m to open man, q to quit)

5

choose option (m to open man, q to quit)

Q

no such option. Try m for man

choose option (m to open man, q to quit)

q

C:\Users\Daniel\Desktop\oop_exercise_05-master\out\build\x64-Debug\oop_exe

rcise_05.exe (процесс 16684) завершил работу с кодом 0.

5. Листинг программы

```
//
//  main.cpp
//  lab6
//  Variant 19
//  М8о-206Б-19
//  Created by Daniel Pivnitskiy on 10.10.2020.
//  github.com/SLAST1
//  Copyright © 2020 Daniel Pivnitskiy. All rights reserved.
//
/*
Разработать шаблоны классов согласно варианту задания.
Параметром шаблона должен являться скалярный тип данных задающий тип данных
```

```
для оси координат.
Классы должны иметь публичные поля. Фигуры являются фигурами вращения, т.е.
равносторонними (кроме трапеции и прямоугольника).
Для хранения координат фигур необходимо использовать шаблон  std::pair.
*/
#include <iostream>
#include <algorithm>
#include "rectangle.h"
#include "containers/queue.h"
#include "allocator.h"

int main() {
    size_t N;
    float S;
    char option = '0';
                                                containers::queue<Rectangle<int>,
allocators::my_allocator<Rectangle<int>, 800>> q;
    Rectangle<int> rect{};
    while (option != 'q') {
        std::cout << "choose option (m to open man, q to quit)" << std::endl;
        std:: cin >> option;
        switch (option) {
            case 'q':
                break;
            case 'm':
                std::cout << "1) push new element into queue\n"
                << "2) insert element into chosen position\n"
                << "3) pop element from the queue\n"
                << "4) delete element from the chosen position\n"
                << "5) print queue\n"
                 << "6) count elements with area less then chosen value\n" <<
std::endl;
                break;
            case '1': {
                        std::cout << "enter rectangle (have to enter dots
consequently): " << std::endl;
                rect = Rectangle<int>(std::cin);
                q.push(rect);
                break;
            }
            case '2': {
                std::cout << "enter position to insert to: ";
                std::cin >> N;
                std::cout << "enter rectangle: ";
                rect = Rectangle<int>(std::cin);
                q.insert_by_number(N, rect);
                break;
            }
            case '3': {
                q.pop();
                break;
```

```
            }
            case '4': {
                std::cout << "enter position to delete: ";
                std::cin >> N;
                q.delete_by_number(N);
                break;
            }
            case '5': {
                    std::for_each(q.begin(), q.end(), [](Rectangle<int> &X) {
X.Printout(std::cout); });
                break;
            }
            case '6': {
                std::cout << "enter max area to search to: ";
                std::cin >> S;
                  std::cout <<"number of elements with value less than " << S
<< " is " << std::count_if(q.begin(), q.end(), [=](Rectangle<int>& X){return
X.Area() < S;}) << std::endl;
                break;
            }
            default:
                std::cout << "no such option. Try m for man" << std::endl;
                break;
        }
    }
    return 0;
}



//
//  rectangle.h
//  lab6
//  Variant 19
//  M8o-206Б-19
//  Created by Daniel Pivnitskiy on 10.10.2020.
//  github.com/SLAST1
//  Copyright © 2020 Daniel Pivnitskiy. All rights reserved.
//
#ifndef OOP_LAB5_RECTANGLE_H
#define OOP_LAB5_RECTANGLE_H

#include "vertex.h"

template <class T>
class Rectangle {
public:
    vertex<T> dots[4];

    explicit Rectangle<T>(std::istream& is) {
        for (auto & dot : dots) {
```

```
            is >> dot;
        }
    }

      Rectangle<T>() = default;

    double Area() {
            double a = sqrt(((dots[1].x - dots[0].x) * (dots[1].x -
dots[0].x)) + ((dots[1].y - dots[0].y) * (dots[1].y - dots[0].y)));
            double b = sqrt(((dots[2].x - dots[1].x) * (dots[2].x -
dots[1].x)) + ((dots[2].y - dots[1].y) * (dots[2].y - dots[1].y)));
            return a * b;
    }

    void Printout(std::ostream& os) {
        for (int i = 0; i < 4; ++i) {
            os << this->dots[i];
            if (i != 3) {
                os << ", ";
            }
        }
        os << std::endl;
    }

    void operator<< (std::ostream& os) {
        for (int i = 0; i < 4; ++i) {
            os << this->dots[i];
            if (i != 3) {
                os << ", ";
            }
        }
    }
};

#endif //OOP_LAB5_RECTANGLE_H



//
//  vertex.h
//  lab6
//  Variant 19
//  M8o-206B-19
//  Created by Daniel Pivnitskiy on 10.10.2020.
//  github.com/SLAST1
//  Copyright © 2020 Daniel Pivnitskiy. All rights reserved.
//
#ifndef OOP_LAB5_VERTEX_H
#define OOP_LAB5_VERTEX_H

#include <iostream>
```

```
#include <type_traits>
#include <cmath>


template<class T>
struct vertex {
    T x;
    T y;
    vertex<T>& operator=(vertex<T> A);
};

template<class T>
std::istream& operator>>(std::istream& is, vertex<T>& p) {
    is >> p.x >> p.y;
    return is;
}

template<class T>
std::ostream& operator<<(std::ostream& os, vertex<T> p) {
    os << '(' << p.x << ' ' << p.y << ')';
    return os;
}

template<class T>
vertex<T> operator+(const vertex<T>& A, const vertex<T>& B) {
    vertex<T> res;
    res.x = A.x + B.x;
    res.y = A.y + B.y;
    return res;
}

template<class T>
vertex<T>& vertex<T>::operator=(const vertex<T> A) {
    this->x = A.x;
    this->y = A.y;
    return *this;
}

template<class T>
vertex<T> operator+=(vertex<T> &A, const vertex<T> &B) {
    A.x += B.x;
    A.y += B.y;
    return A;
}

template<class T>
vertex<T> operator/=(vertex<T>& A, const double B) {
    A.x /= B;
    A.y /= B;
}
```

```
template<class T>
double vert_length(vertex<T>& A, vertex<T>& B) {
    double res = sqrt( pow(B.x - A.x, 2) + pow(B.y - A.y, 2) );
    return res;
}

template<class T>
struct is_vertex : std::false_type {};

template<class T>
struct is_vertex<vertex<T>> : std::true_type {};

#endif //OOP_LAB5_VERTEX_H
```

```
//
//  allocator.h
//  lab6
//  Variant 19
//  М8о-206Б-19
//  Created by Daniel Pivnitskiy on 10.10.2020.
//  github.com/SLAST1
//  Copyright © 2020 Daniel Pivnitskiy. All rights reserved.
//
#ifndef OOP_EXERCISE_05_ALLOCATOR_H_
#define OOP_EXERCISE_05_ALLOCATOR_H_

#include <cstdlib>
#include <iostream>
#include <type_traits>

#include "containers/stack.h"

namespace allocators {

    template<class T, size_t a_size>
    struct my_allocator {
        using value_type = T;
        using size_type = std::size_t;
        using difference_type = std::ptrdiff_t;
        using is_always_equal = std::false_type;

        template<class U>
        struct rebind {
            using other = my_allocator<U, a_size>;
        };

        my_allocator() :
            begin(new char[a_size]),
            end(begin + a_size),
```

```
            tail(begin)
        {}

        my_allocator(const my_allocator&) = delete;
        my_allocator(my_allocator&&) = delete;

        ~my_allocator() {
            delete[] begin;
        }

        T* allocate(std::size_t n);
        void deallocate(T* ptr, std::size_t n);

    private:
        char* begin;
        char* end;
        char* tail;
        containers::stack<char*> free_blocks;
    };

    template<class T, size_t a_size>
    T* my_allocator<T, a_size>::allocate(std::size_t n) {
        if (n != 1) {
            throw std::logic_error("can`t allocate arrays");
        }
        if (size_t(end - tail) < sizeof(T)) {
            if (free_blocks.Size()) {
                auto it = free_blocks.begin();
                char* ptr = *it;
                free_blocks.pop();
                return reinterpret_cast<T*>(ptr);
            }
            throw std::bad_alloc();
        }
        T* result = reinterpret_cast<T*>(tail);
        tail += sizeof(T);
        return result;
    }

    template<class T, size_t a_size>
    void my_allocator<T, a_size>::deallocate(T* ptr, std::size_t n) {
        if (n != 1) {
            throw std::logic_error("can`t deallocate arrays");
        }
        if (ptr == nullptr) {
            return;
        }
        free_blocks.push(reinterpret_cast<char*>(ptr));
    }
}
```

```
#endif


//
//  stack.h
//  lab6
//  Variant 19
//  M8o-206B-19
//  Created by Daniel Pivnitskiy on 10.10.2020.
//  github.com/SLAST1
//  Copyright © 2020 Daniel Pivnitskiy. All rights reserved.
//
#ifndef OOP_EXERCISE_06_STACK_H
#define OOP_EXERCISE_06_STACK_H

#include <iterator>
#include <memory>
#include <algorithm>

namespace containers {


    template<class T, class Allocator = std::allocator<T>>
    class stack {
    private:
        struct element;
        size_t size = 0;
    public:
        stack() = default;

        class forward_iterator {
        public:
            using value_type = T;
            using reference = T&;
            using pointer = T*;
            using difference_type = std::ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;
            explicit forward_iterator(element* ptr);
            T& operator*();
            forward_iterator& operator++();
            forward_iterator operator++(int);
            bool operator== (const forward_iterator& other) const;
            bool operator!= (const forward_iterator& other) const;
        private:
            element* it_ptr;
            friend stack;
        };

        forward_iterator begin();
        forward_iterator end();
        void push(const T& value);
```

```
        T& top();
        void pop();
        void delete_by_it(forward_iterator d_it);
        void delete_by_number(size_t N);
        void insert_by_it(forward_iterator ins_it, T& value);
        void insert_by_number(size_t N, T& value);
        size_t Size();
    private:
                    using allocator_type = typename Allocator::template
rebind<element>::other;

        struct deleter {
            deleter(allocator_type* allocator) : allocator_(allocator) {}

            void operator() (element* ptr) {
                if (ptr != nullptr) {

std::allocator_traits<allocator_type>::destroy(*allocator_, ptr);
                    allocator_->deallocate(ptr, 1);
                }
            }

        private:
            allocator_type* allocator_;
        };

        struct element {
            T value;
                    std::unique_ptr<element, deleter> next_element{ nullptr,
deleter{nullptr} };
            element(const T& value_) : value(value_) {}
            forward_iterator next();
        };
        allocator_type allocator_{};
        std::unique_ptr<element, deleter> first{ nullptr, deleter{nullptr} };
    };

    template<class T, class Allocator>
              typename   stack<T,   Allocator>::forward_iterator   stack<T,
Allocator>::begin() {
        return forward_iterator(first.get());
    }

    template<class T, class Allocator>
    typename stack<T, Allocator>::forward_iterator stack<T, Allocator>::end()
{
        return forward_iterator(nullptr);
    }

    template<class T, class Allocator>
    void stack<T, Allocator>::push(const T& value) {
```

```
        element* tmp = this->allocator_.allocate(1);
           std::allocator_traits<allocator_type>::construct(this->allocator_,
tmp, value);
        if (first == nullptr) {
                    first = std::unique_ptr<element, deleter>(tmp, deleter{
&this->allocator_ });
        }
        else {
            std::swap(tmp->next_element, first);
            first = std::move(std::unique_ptr<element, deleter>(tmp, deleter{
&this->allocator_ }));
        }
        size++;
    }

    template<class T, class Allocator>
    void stack<T, Allocator>::pop() {
        if (size == 0) {
            throw std::logic_error("stack is empty");
        }
//                                    element*    tmp    =    std::unique_ptr
<element,deleter>(first->next_element);
                                   std::unique_ptr<element,deleter>    tmp    =
std::move(first->next_element);
        first = std::move(tmp);
        size--;
    }

    template<class T, class Allocator>
    T& stack<T, Allocator>::top() {
        if (size == 0) {
            throw std::logic_error("stack is empty");
        }
        return first->value;
    }

    template<class T, class Allocator>
    size_t stack<T, Allocator>::Size() {
        return size;
    }

    template<class T, class Allocator>
              void   stack<T,   Allocator>::delete_by_it(containers::stack<T,
Allocator>::forward_iterator d_it) {
        forward_iterator i = this->begin(), end = this->end();
        if (d_it == end) throw std::logic_error("out of borders");
        if (d_it == this->begin()) {
            this->pop();
            return;
        }
        while ((i.it_ptr != nullptr) && (i.it_ptr->next() != d_it)) {
```

```
            ++i;
        }
        if (i.it_ptr == nullptr) throw std::logic_error("out of borders");
        i.it_ptr->next_element = std::move(d_it.it_ptr->next_element);
        size--;
    }

    template<class T, class Allocator>
    void stack<T, Allocator>::delete_by_number(size_t N) {
        forward_iterator it = this->begin();
        for (size_t i = 1; i <= N; ++i) {
            if (i == N) break;
            ++it;
        }
        this->delete_by_it(it);
    }

    template<class T, class Allocator>
              void   stack<T,   Allocator>::insert_by_it(containers::stack<T,
Allocator>::forward_iterator ins_it, T& value) {
        element* tmp = this->allocator_.allocate(1);
           std::allocator_traits<allocator_type>::construct(this->allocator_,
tmp, value);
        forward_iterator i = this->begin();
        if (ins_it == this->begin()) {
            tmp->next_element = std::move(first);
            first = std::move(std::unique_ptr<element, deleter>(tmp, deleter{
&this->allocator_ }));
            size++;
            return;
        }
        while ((i.it_ptr != nullptr) && (i.it_ptr->next() != ins_it)) {
            i++;
        }
        if (i.it_ptr == nullptr) throw std::logic_error("out of borders");
        tmp->next_element = std::move(i.it_ptr->next_element);
                i.it_ptr->next_element  =  std::move(std::unique_ptr<element,
deleter>(tmp, deleter{ &this->allocator_ }));
        size++;
    }
    template<class T, class Allocator>
    void stack<T, Allocator>::insert_by_number(size_t N, T& value) {
        forward_iterator it = this->begin();
        for (size_t i = 1; i <= N; ++i) {
            if (i == N) break;
            ++it;
        }
        this->insert_by_it(it, value);
    }

    template<class T, class Allocator>
```

```
            typename    stack<T,    Allocator>::forward_iterator    stack<T,
Allocator>::element::next() {
        return forward_iterator(this->next_element.get());
    }

    template<class T, class Allocator>
                                                                        stack<T,
Allocator>::forward_iterator::forward_iterator(containers::stack<T,
Allocator>::element* ptr) {
        it_ptr = ptr;
    }

    template<class T, class Allocator>
    T& stack<T, Allocator>::forward_iterator::operator*() {
        return this->it_ptr->value;
    }

    template<class T, class Allocator>
             typename    stack<T,    Allocator>::forward_iterator&    stack<T,
Allocator>::forward_iterator::operator++() {
             if  (it_ptr  ==  nullptr)  throw  std::logic_error("out  of  stack
borders");
        *this = it_ptr->next();
        return *this;
    }

    template<class T, class Allocator>
             typename    stack<T,    Allocator>::forward_iterator    stack<T,
Allocator>::forward_iterator::operator++(int) {
        forward_iterator old = *this;
        ++* this;
        return old;
    }

    template<class T, class Allocator>
            bool    stack<T,    Allocator>::forward_iterator::operator==(const
forward_iterator& other) const {
        return it_ptr == other.it_ptr;
    }

    template<class T, class Allocator>
            bool    stack<T,    Allocator>::forward_iterator::operator!=(const
forward_iterator& other) const {
        return it_ptr != other.it_ptr;
    }
}


#endif
```

```
//
//  queue.h
//  lab6
//  Variant 19
//  М8о-206Б-19
//  Created by Daniel Pivnitskiy on 10.10.2020.
//  github.com/SLAST1
//  Copyright © 2020 Daniel Pivnitskiy. All rights reserved.
//
#ifndef OOP_EXERCISE_05_QUEUE_H
#define OOP_EXERCISE_05_QUEUE_H

#include <iterator>
#include <memory>
#include <algorithm>

namespace containers {

    template<class T, class Allocator = std::allocator<T>>
    class queue {
    private:
        struct element;
        size_t size = 0;
    public:
        queue() = default;

        class forward_iterator {
        public:
            using value_type = T;
            using reference = T &;
            using pointer = T *;
            using difference_type = std::ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;

            explicit forward_iterator(element *ptr);

            T &operator*();

            forward_iterator &operator++();

            forward_iterator operator++(int);

            bool operator==(const forward_iterator &other) const;

            bool operator!=(const forward_iterator &other) const;

        private:
            element *it_ptr;
            friend queue;
        };
```

```
        forward_iterator begin();

        forward_iterator end();

        void push(const T &value);

        T &top();

        void pop();

        void delete_by_it(forward_iterator d_it);

        void delete_by_number(size_t N);

        void insert_by_it(forward_iterator ins_it, T &value);

        void insert_by_number(size_t N, T &value);

        size_t Size();

    private:
                    using allocator_type = typename Allocator::template
rebind<element>::other;

        struct deleter {
            deleter(allocator_type *allocator) : allocator_(allocator) {}

            void operator()(element *ptr) {
                if (ptr != nullptr) {

std::allocator_traits<allocator_type>::destroy(*allocator_, ptr);
                    allocator_->deallocate(ptr, 1);
                }
            }

        private:
            allocator_type *allocator_;
        };

        struct element {
            T value;
                     std::unique_ptr<element, deleter> next_element{nullptr,
deleter{nullptr}};

            element(const T &value_) : value(value_) {}

            forward_iterator next();
        };

          static std::unique_ptr<element> push_impl(std::unique_ptr<element>
```

```
cur, const T &value);

        //std::unique_ptr<element> first = nullptr;
        allocator_type allocator_{};
        std::unique_ptr<element, deleter> first{nullptr, deleter{nullptr}};
    };

    template<class T, class Allocator>
    void queue<T, Allocator>::push(const T &value) {
        element *tmp = this->allocator_.allocate(1);
           std::allocator_traits<allocator_type>::construct(this->allocator_,
tmp, value);
        if (first == nullptr) {
                            first  =  std::unique_ptr<element,  deleter>(tmp,
deleter{&this->allocator_});
        } else {
            std::swap(tmp->next_element, first);
                   first  =  std::move(std::unique_ptr<element,  deleter>(tmp,
deleter{&this->allocator_}));
        }
        size++;
    }

    template<class T, class Allocator>
              typename   queue<T,   Allocator>::forward_iterator   queue<T,
Allocator>::begin() {
        return forward_iterator(first.get());
    }

    template<class T, class Allocator>
    typename queue<T, Allocator>::forward_iterator queue<T, Allocator>::end()
{
        return forward_iterator(nullptr);
    }

    template<class T, class Allocator>
    void queue<T, Allocator>::pop() {
        if (size == 0) {
            throw std::logic_error("queue is empty");
        }
        this->first = std::move(this->begin().it_ptr->next_element);
    size--;
    }

    template<class T, class Allocator>
    T &queue<T, Allocator>::top() {
        if (size == 0) {
            throw std::logic_error("stack is empty");
        }
                                  auto    tmp    =    std::unique_ptr<element,
deleter>(std::move(first->next_element));
```

```
        first = std::move(tmp);
        size--;
    }

    template<class T, class Allocator>
    size_t queue<T, Allocator>::Size() {
        return size;
    }

    template<class T, class Allocator>
    std::unique_ptr<typename queue<T, Allocator>::element>
      queue<T, Allocator>::push_impl(std::unique_ptr<element> cur, const T
&value) {
        if (cur != nullptr) {
                cur->next_element = push_impl(std::move(cur->next_element),
value);
            return cur;
        }
        return std::unique_ptr<element>(new element{value});
    }

    template<class T, class Allocator>
             void   queue<T,   Allocator>::delete_by_it(containers::queue<T,
Allocator>::forward_iterator d_it) {
        forward_iterator i = this->begin(), end = this->end();
        if (d_it == end) throw std::logic_error("out of borders");
        if (d_it == this->begin()) {
            this->pop();
            return;
        }
        while ((i.it_ptr != nullptr) && (i.it_ptr->next() != d_it)) {
            ++i;
        }
        if (i.it_ptr == nullptr) throw std::logic_error("out of borders");
        i.it_ptr->next_element = std::move(d_it.it_ptr->next_element);
        size--;
    }

    template<class T, class Allocator>
    void queue<T, Allocator>::delete_by_number(size_t N) {
        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i, ++it) {}
        this->delete_by_it(it);
    }

    template<class T, class Allocator>
             void   queue<T,   Allocator>::insert_by_it(containers::queue<T,
Allocator>::forward_iterator ins_it, T &value) {
        element *tmp = this->allocator_.allocate(1);
           std::allocator_traits<allocator_type>::construct(this->allocator_,
tmp, value);
```

```
        forward_iterator i = this->begin();
        if (ins_it == this->begin()) {
            tmp->next_element = std::move(first);
                   first = std::move(std::unique_ptr<element, deleter>(tmp,
deleter{&this->allocator_}));
            size++;
            return;
        }
        while ((i.it_ptr != nullptr) && (i.it_ptr->next() != ins_it)) {
            i++;
        }
        if (i.it_ptr == nullptr) throw std::logic_error("out of borders");
        tmp->next_element = std::move(i.it_ptr->next_element);
                i.it_ptr->next_element = std::move(std::unique_ptr<element,
deleter>(tmp, deleter{&this->allocator_}));
        size++;
    }

    template<class T, class Allocator>
    void queue<T, Allocator>::insert_by_number(size_t N, T &value) {
        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
            ++it;
        }
        this->insert_by_it(it, value);
    }

    template<class T, class Allocator>
               typename queue<T, Allocator>::forward_iterator queue<T,
Allocator>::element::next() {
        return forward_iterator(this->next_element.get());
    }

    template<class T, class Allocator>
                                                                    queue<T,
Allocator>::forward_iterator::forward_iterator(containers::queue<T,
Allocator>::element *ptr) {
        it_ptr = ptr;
    }

    template<class T, class Allocator>
    T &queue<T, Allocator>::forward_iterator::operator*() {
        return this->it_ptr->value;
    }

    template<class T, class Allocator>
               typename queue<T, Allocator>::forward_iterator &queue<T,
Allocator>::forward_iterator::operator++() {
               if (it_ptr == nullptr) throw std::logic_error("out of queue
borders");
        *this = it_ptr->next();
```

```
        return *this;
    }

    template<class T, class Allocator>
              typename    queue<T,    Allocator>::forward_iterator    queue<T,
Allocator>::forward_iterator::operator++(int) {
        forward_iterator old = *this;
        ++*this;
        return old;
    }

    template<class T, class Allocator>
             bool    queue<T,    Allocator>::forward_iterator::operator==(const
forward_iterator &other) const {
        return it_ptr == other.it_ptr;
    }

    template<class T, class Allocator>
             bool    queue<T,    Allocator>::forward_iterator::operator!=(const
forward_iterator &other) const {
        return it_ptr != other.it_ptr;
    }
}

#endif
```

6. Выводы

В ходе данной лабораторной работы были получены навыки работы с аллокаторами. Аллокаторы позволяют ускорить быстродействие программ, сократив количество системных вызовов, а также усилить контроль над менеджментом памяти

7. Список литературы

1.Руководство по написанию кода на C++ [Электронный ресурс].
URL:https://metanit.com/cpp/tutorial/
Дата обращения: 10.09.2019

2.Документация по C++ [Электронный ресурс]. URL:
https://docs.microsoft.com/ru-ru/cpp
Дата обращения 12.09.2019